Université Claude Bernard Lyon 1

Rapport de projet

Gestion de Projet et Génie Logiciel

Par BESSAADI Ilyess & BEN OTHMAN Samy

Table des matières

Présentation du projet

Dans le cadre de l'Unité d'enseignement « **Gestion de Projet et Génie Logiciel** », il nous a été demandé de développer en **Java** un simulateur prototype de l'application **StopCovid**. Cette dernière permet de prévenir les utilisateurs qui ont été à proximité d'une personne testée positive au Covid, afin que celles-ci puissent être prises en charge le plus tôt possible, le tout sans jamais sacrifier leurs libertés individuelles. Pour cela, les utilisateurs peuvent se déclarer positif, et l'application se chargera d'alerter les utilisateurs avec qui ils ont été en contact. Notre projet simulera donc la rencontre entre plusieurs utilisateurs afin de voir ce qu'il se passe si l'une de ces personnes est infectée.

Pour commencer une simulation, il faut d'abord lancer le serveur afin d'accéder à deux fonctionnalités : sa configuration, et la création de clients. Ensuite, le serveur pourra être arrêté, entrainant la suppression des clients, puis relancer par la suite si l'on veut recommencer une simulation.

La configuration du serveur peut être faite à tout moment, et s'adaptera en fonction de l'état de la simulation. Elle ne possède qu'un seul paramètre, le niveau de prudence, qui impacte la sensibilité de l'évaluation des risques. Lorsqu'il est faible, seuls les utilisateurs à risque élevés sont alertés, et inversement s'il est plus élevé. En effet, contrairement à ce qui est initialement prévu par le sujet, ici, les risques sont quantifiés, et tout contact, même à faible risque est pris en compte. C'est pour cette raison que les clients doivent envoyer leurs contacts même s'ils ne sont pas malades.

La création de clients est faite dynamiquement durant la simulation, c'est pourquoi le constructeur de clients initialement prévu par le sujet en serrait inutile et n'est donc pas implémenter.

Lors de la création d'un client, son panneau de configuration est aussi créé, où l'on y retrouve plusieurs éléments.

Tout d'abord, l'identifiant et le statut du client y sont affichés, et mis à jour.

Le statut est modifié lorsqu'un client se déclare infecté ou que le serveur le déclare à risque, ou sans risque. L'identifiant d'un client est éphémère, et donc renouvelé périodiquement, afin de garantir à chaque client un anonymat envers les autres utilisateurs.

Ensuite, la stratégie d'envoi des contacts du client peut être configurée pour qu'il envoie tous ses contacts, ses contacts répétés ou fréquents.

Le sujet prévoyait initialement l'ajout d'une autre stratégie, pour n'envoyer que les 10 contacts les plus fréquents, mais cette fonctionnalité n'a pas été implémentée, puisqu'elle perd son sens avec l'implémentation d'identifiant éphémère.

Ensuite, il y a la simulation de rencontre avec d'autres utilisateurs. Puis, le bouton d'ouverture de l'application simulé du client où il peut se déclarer infecté.

Enfin, il y a le bouton de suppression du client, qui entraînera aussi la fermeture de son application simulée s'il est ouvert, ainsi que de son panneau de configuration.

Durant la simulation, en plus de l'évaluation dynamique des risques des clients en fonction de leurs contacts, chaque contact d'un client est supprimé après un certain temps représentant une quatorzaine, en remplacement de la suppression manuel de contacts initialement prévu par le sujet. Pour aller de pair avec cette fonctionnalité, l'infection d'un client possède aussi une durée de vie représentant une quatorzaine.

Pour fermer l'application de simulation, il suffit de fermer la fenêtre principale, et tout s'arrêtera, peu importe l'état de la simulation ou si d'autres fenêtres sont ouvertes.

Design patterns

L'utilisation de patron de conception et d'architecture est une partie importante du projet. Nous allons donc vous présenter nos choix.

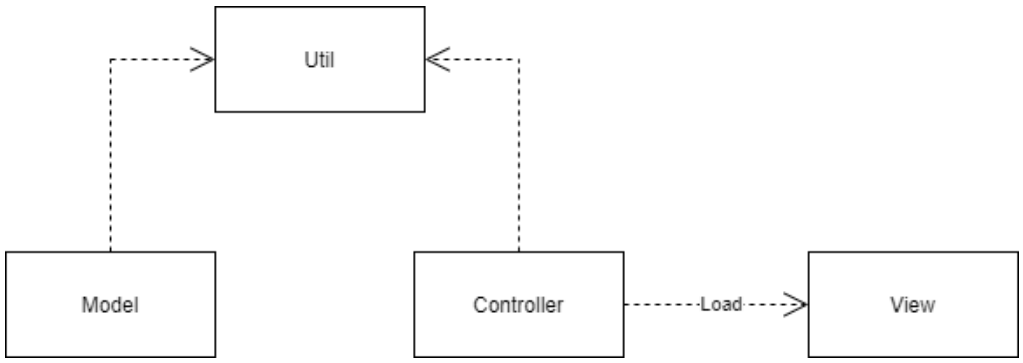
Patron d'architecture : MVC

Le patron d'architecture **MVC** permet d'organiser et de séparer le développement d'une application en trois modules ayant trois responsabilités différentes : le modèle (**model**), la vue (**view**) et le contrôleur (**controller**).

Les modèles contiennent les données, les vues contiennent l'interface graphique, et les contrôleurs contiennent la logique des actions utilisateur.

Dans notre cas, nous utilisons **JavaFXML**, une vue n'est donc qu'un fichier **XML**, qui sera chargé en tant qu'objet **JavaFX** auquel on attribuera un contrôleur qui aura la responsabilité de sa modification durant l'exécution du programme. Ici, une partie des responsabilités de la vue ont donc été délégué au contrôleur.

En plus de ces trois modules, nous en avons aussi un autre nommé « **util** » contenant des modules qui ne dépendent de rien d'autre dans le projet, et qui ne sont pas spécifique à un des trois module MVC. C'est par exemple le cas de nos énumérateurs et de notre système d'événement.



Patron de conception : Observateur

Si deux objets ont besoin de communiquer entre eux, une première solution est que chaque objet possède une référence de l'autre. Mais cela créer un fort couplage qui est à éviter. C'est la raison pour laquelle nous utilisons le patron de conception « **observateur** », qui permet une communication sans dépendance, et limite donc le couplage aux observables.

Ce patron de conception peut être utilisé avec différentes solutions, et nous avons choisi d'utiliser un système d'événement (que l'on a nous-même implémenté).

Un événement peut recevoir des abonnements de références de fonction (dites « **fonction de rappel** ») qu'il pourra par la suite exécuter, par exemple pour prévenir de certaines modifications. Le principe d'un événement est donc de pouvoir communiquer avec d'autres objets sans dépendre d'eux, puisque ce sont eux qui y abonnent leur fonction de rappel.

Dans ce cas, on dit que les objets possédant des événements sont des « observables », et ceux qui y abonnent leurs fonctions de rappel sont des « observateurs ».

Dans notre projet, nous avons deux cas d'utilisation.

Premièrement, des objets du contrôleur doivent communiquer avec d'autres du modèle, et inversement, pour par exemple mettre à jour la vue. Pour avoir un faible couplage et pour respecter le patron MVC qui impose un modèle indépendant, ces contrôleurs sont des observateurs et ces modèles sont des observables.

Deuxièmement, lorsqu'un client change de statut, ses contacts doivent être prévenus. Pour éviter de tout vérifier à chaque changement de statut ou d'avoir énormément de dépendances circulaires emmêlé, le gestionnaire d'un client est observateur de l'état de ses contacts, et le serveur est observateur des états de tous les clients, qui sont donc des observables.

Figure 2 : Premier cas

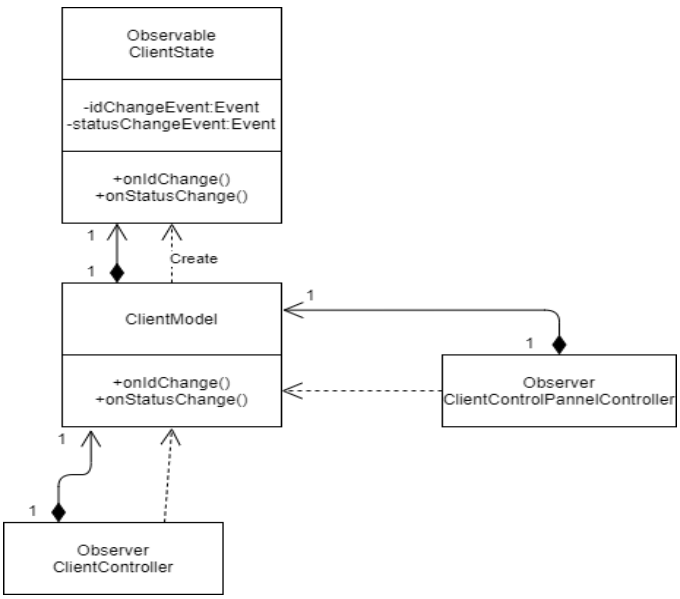
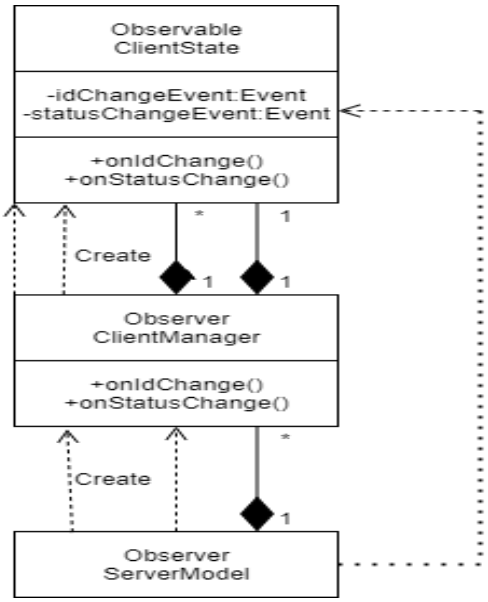


Figure 1 : Deuxième cas



Patron de conception : Fonction de rappel

Une **fonction de rappel** est une référence de fonction, qui peut donc être passée en argument d'une autre fonction.

Nous utilisons ce patron de conception puisqu'il est nécessaire aux événements que nous utilisons lors de l'application du patron de conception « observateur » comme mentionné ci-dessus, mais aussi lors de l'utilisation des éléments JavaFX qui en utilisent.

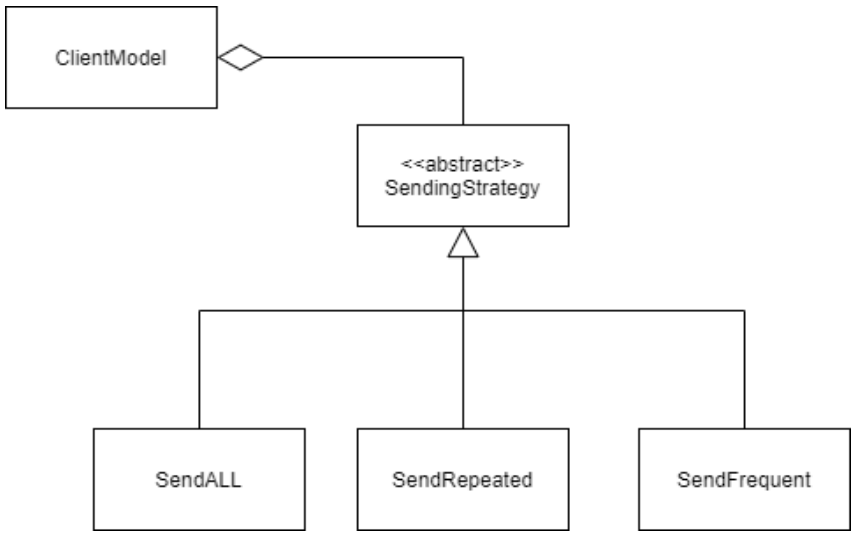
Patron de conception : Stratégie

Comme présenter lors de la première partie, nos clients utilisent différentes stratégies d'envoi de contacts, qui peuvent être configuré durant l'exécution du programme. Selon la stratégie d'envoi, le client doit donc agir différemment.

Une première solution serrait que le client possède en attribut une variable simple tel qu'un entier ou un énumérateur, puis lors de l'ajout d'un contact, le client vérifiera s'il doit envoyer le contact ou non selon la valeur de la variable, et agira en conséquence. Mais cette solution n'est pas très extensible, puisqu'en effet, si l'on veut ajouter une nouvelle stratégie, il faudra directement modifier le code du client.

C'est pour cela que nous utilisons le patron de conception « **stratégie** », qui est très utile dans ces cas-là. Son application est dans la mise en place d'une famille de classes indépendantes à qui on déléguera des responsabilités, et qui effectuerons souvent leurs tâches en fonction d'un contexte donnée. Cette solution est bien plus extensible, puisque pour gérer un nouveau comportement, il n'y a qu'à créer une nouvelle classe de cette famille, avec un comportement différent.

Ici, nous avons donc la famille de classes des stratégies d'envoi qui ne possèdent qu'une seule méthode puisqu'on leur a confié l'unique responsabilité d'informer quant à l'envoi d'un contact donné.



Ethique

Afin d'assurer un bon fonctionnement, l'application doit récupérer des données afin d'identifier les personnes atteintes du coronavirus et de prévenir ceux qui ont été en contact avec ces malades.

Pour atteindre ce but, tout en respectant la vie privée des utilisateurs, deux critères doivent être respectés. Premièrement, les autorités ne doivent pas pouvoir récupérer plus d'informations sur un utilisateur que ce qu'il choisit d'envoyer à l'application. Deuxièmement, les utilisateurs doivent être assuré d'un anonymat envers les autres utilisateurs.

Pour le développement de l'application StopCovid, deux solutions ont été proposées. Les deux solutions proposent d'identifier les utilisateurs avec un identifiant éphémère, pour assurer leur anonymat.

La première solution proposée par Google et Apple met l'utilisateur au centre. L'utilisateur doit régulièrement consulter une liste tenue par les autorités afin de savoir s'il est à risque ou non. Cette solution empêche donc toute collecte de données possible, mais a en effet ses limites. Premièrement, bien qu'identifié par un identifiant éphémère, envoyer ses identifiants vers un registre consultable par ses contacts peut potentiellement permettre d'être identifié comme malade auprès de ces derniers, et réduit donc considérablement l'anonymat. Aussi, puisque l'utilisateur a un contrôle total de ses données, il peut choisir de ne pas se déclarer positif ou à risque, même s'il l'est. Dans le cas où une partie des utilisateurs ne communiquent pas leurs données, cette solution devient donc moins fiable et moins intéressante.

La deuxième solution proposée par l'INRIA que l'on appelle la méthode ROBERT, donne plus de pouvoir à l'état. Ce dernier s'occupe de notifier les personnes entrées en contact avec une personne malade sans jamais révéler la source de l'infection. Chaque utilisateur est identifié par une clé privée ce qui minimise les chances de pouvoir identifier une personne malade. Toutefois cette solution n'est pas miracle. Bien que les malades soient protégés, c'est l'autorité qui a les pleins pouvoirs concernant la divulgation de la maladie. Si cette dernière est malhonnête elle peut garder les données ou retracer les malades et ainsi enfreindre le respect de la vie privée tout en ayant main mise sur les décisions de diffusion des informations de la personne malade.

De ces deux solutions nous pouvons en extraire que la première voit ses utilisateurs comme des personnes responsables qui vont tout le temps informer s'ils sont malades ou non. L'autorité ne joue pas un rôle important puisqu'il ne sert que de relais. Les utilisateurs peuvent cependant identifier la personne porteuse du virus. Dans le deuxième cas on suppose que l'autorité est bienveillante et que ce sont les utilisateurs qu'il faut protéger des autres utilisateurs en ne leur divulguant pas d'informations sur le malade qui les a contaminés. Il faut donc établir une relation de confiance entre l'autorité qui collecte les données et l'utilisateur. Cela pose des problèmes de respect de la vie privée puisqu'on sait que l'autorité n'est jamais totalement bienveillante en ce qui concerne la récolte de données.

Ces deux solutions ont tout de même des limites. En effet, si une personne malade n'a pas de téléphone alors le principe de l'application n'a plus aucun sens. Ces deux procédés sont donc difficilement exportables dans les pays pauvres par exemple. Pour pallier ce problème, ces systèmes de localisation des malades vont être implémenté sur des bracelets, offrant donc à tous la possibilité de se signaler comme malade. Cependant cette solution a encore des limites puisqu'il n'y a aucun moyen de contacter une personne qui est entrée en contact avec un malade.

Il est donc difficile de trouver une solution parfaite mais nous avons quand même essayé.

Pour notre projet, nous avons donc décidé de trouver un compromis entre avoir une application fiable et respecter la vie privée. En effet, chaque utilisateur est référencé par une clé générée aléatoirement. Le serveur n'a donc pas la possibilité de remonter aux personnes malades puisqu'il ne stocke aucune information les concernant. Il n'a par ailleurs pas la possibilité de déclarer une personne malade sans son consentement. Le serveur ne contrôle donc pas tout et respecte la vie privée de nos utilisateurs tout en assurant un service qualitatif.

Tests

Concernant les tests unitaires, nous avons décidé d'effectuer chaque test de chaque classe dans des fichiers séparés. Cela permet une meilleure compréhension de ces derniers. Pour les fonctions publiques nous nous sommes penchés vers des assertions classiques avec Junit mais nous avons aussi utilisé d'autres outils tel que Mockito. En effet, pour tester certaines fonctions il faut tout instancier afin d'avoir un scénario cohérent. Mockito est donc un outil qui permet de fabriquer des faux objets (des mocks) afin de nous éviter de multiples instanciations.

En ce qui concerne les fonctions privées, nous ne pouvons pas les tester avec Junit mais nous avons décidé de les tester à l'intérieur même de la fonction (avec des blocks try/catch par exemple).